Operationalization of a graphical knowledge representation language

B. Charreton, J-L Ermine

EKAW'96, European Knowledge Acquisition Workshop, Nottingham, UK, May 1996, Poster Session, pp 2- 12

And also: «From knowledge specification to executable specification, Knowledge Engineering and Modelling Languages

KEML'96, Paris, 15-16 January 1996

French version unpublished

Operationalization of a graphical knowledge representation language

Boris Charreton*, Jean-Louis Ermine *

** CEA/DIST*
Section Méthodes et Technologies de l'Information
Groupe Gestion des Connaissances
Centre d'Etudes de Saclay
91191 Gif sur Yvette CEDEX
tel: 69 08 61 14 / Fax: 69 08 26 69 /e-mail: bcharret@instn.saclay.cea.fr

ABSTRACT: MOISE is a knowledge engineering methodology which includes a knowledge specification stage which separates static knowledge from dynamic knowledge. This stage integrates a graphical knowledge specification language (KRL) that combines a static specification language (semantic networks) and a dynamic specification language (task language). The modelling language KRL is the source language describing knowledge which becomes available for consultation. Some additional tools transform source graphical knowledge descriptions into different target languages: textual descriptions (word), hypertextual descriptions (html) and executable descriptions (SPIRAL). The paper deals with the latter tool. It presents the KRL itself (knowledge-level) and sketches the design model (symbol-level) that corresponds to its executable form and that is implemented in the SPIRAL object-oriented language.

KEYWORDS: knowledge representation language, graphical language, knowledge engineering, object-oriented language, static knowledge, dynamic knowledge, task, semantic networks.

1. Introduction

What is MOISE?
MOISE is a knowledge engineering methodology developed by J-L. Ermine [Ermi93]. It was created around 1989 for a research use whereas KOD [Voge89] or KADS [Breu94] methodologies are intended for industrial applications. Nevertheless it has been successfully applied in about 30 industrial projects, in a large variety of fields such as technical diagnosis [Ermi91] or power plant systems [Benz95]. Some software tools (not commercialized) were also developed to support the methodology. MOISE is now evolving to be integrated in a more general knowledge engineering methodology whose issue goes past and includes the cognitive approach [Brun94].

Knowledge-level versus symbol-level
MOISE includes a knowledge specification phase that aims at building a formalized model of the whole knowledge contributing to the problem at hand. This is a modelling phase at the knowledge-level, as well as the conceptual model construction of KADS methodology is [Wiel92]. In respect of Newel statements, knowledge-level and symbol-level are distinguished [Newe82].

Knowledge-level for a better understanding of expert system knowledge? Or of human expert knowledge?
Two motivations underlie the knowledge level theory [Stee89]. First, it offers "better ways to understand the knowledge contained in expert systems and hence understand why they work or not work." Secondly, it proposes "ways to explicate knowledge engineering skills so that we can teach them". These are not contradictory; nevertheless we favour the last one. That is why a graphical language which is simple and intuitive was chosen to obtain knowledge descriptions that are used as descriptions of reference.

A graphical knowledge representation language
This knowledge specification (or conceptual modelling) is the more often used phase of the method. It separates *static* knowledge that describes domain knowledge, from *dynamic* knowledge that describes the expert strategy by identifying the tasks manipulating domain knowledge objects. Semantic links are defined between those objects, between those tasks and between each others. On one hand an object is represented by a semantic networks language which defines what we call a *concept*, on the other hand a task is represented by a task

language with a graphical representation. Now we shall refer to the Knowledge Representation Language (KRL) as the combination of the semantic networks language and the task language.

KRL descriptions: the descriptions of reference!

The KRL modelling language is the source language describing knowledge which becomes available for consultation. In addition some soft tools, combined with the methodology, transform source graphical knowledge descriptions into different target languages (Figure 1): textual descriptions (word), hypertextual descriptions (html) and executable descriptions. The first transformation leads to report-like document that is a very useful form as it is showed by the report-centred expert system KNACK [Derm88]. The second one is being studied and will allow a hypertextual navigation among graphical knowledge models [Chai95] and can therefore produce a very attractive means of consultation. The latter research direction (operationalization) could help us in prototyping, validating and verifying conceptual models and is the paper centre of interest.

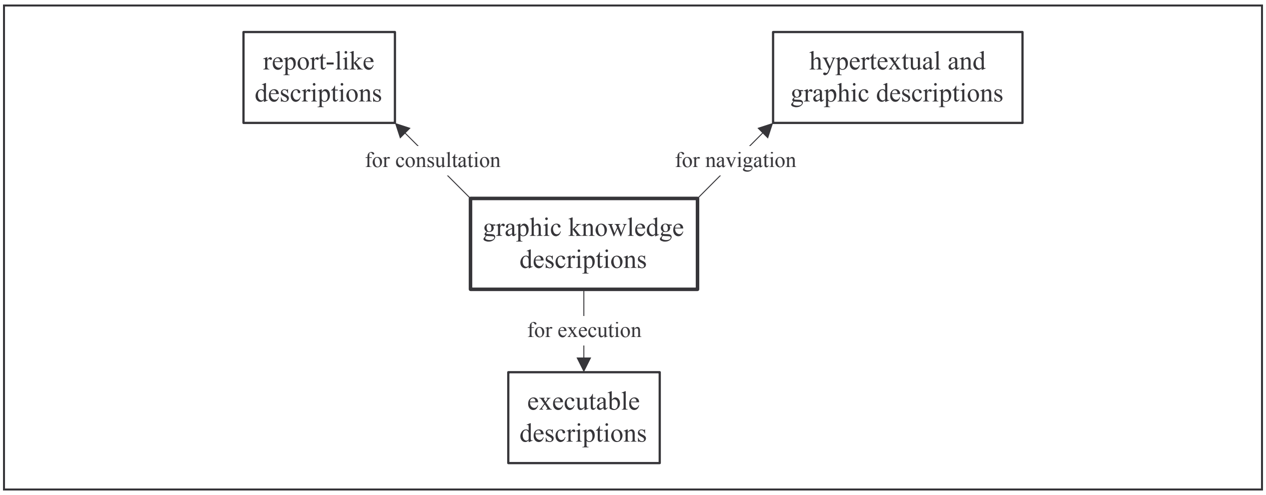

Figure 1: KRL descriptions: central descriptions

Section 2 presents the graphical KRL used to specify static and dynamic knowledge and section 3 details the design model that permits the transformation from graphical knowledge descriptions to executable descriptions. Along the paper and as much as possible our explanations are illustrated by applying them to the Sisyphus II example. For memory Sisyphus II problem [Yost92] is an elevator system configuring problem derived from the source code for the VT elevator configuration expert system [Marc88]. This project aims at comparing different approaches to aspects of knowledge engineering and is complete and complex enough to avoid toy examples.

2. Knowledge representation language

MOISE approach covers all steps from initial knowledge acquisition to design and implementation. Acquisition activity or expert interview techniques are not the paper centre of interest which just focuses on the KRL integrating both static and dynamic languages.

When a KRL is provided to a human expert who initially doesn’t know it, it implies it must rapidly fit in with him. It must be easy and quick to understand. It must own the sufficient and necessary modelling primitives that are useful and suitable for an expert. The model of expertise constructed by the expert should constitute an important means of communication; therefore the KRL should get expressiveness and preciseness properties. According to these reasons a graphical KRL was defined. It combines a semantic networks language to represent static knowledge and a task language to represent dynamic knowledge.

2.1 Semantic networks

Semantic networks are a well known language used to represent static knowledge since they were used by the scholastic logicians in the middle ages. Nets are useful for representing and structuring domain knowledge and more simply for « making our ideas clear » [Peir78]. They are more understandable than first-order predicate calculus [Sowa91] [Bläs89]. Semantic networks terminology and notation vary widely and our version is now presented. A net is a pattern of interconnected nodes and arcs. Nodes represent concepts of entities and terminal nodes may be value sets (Real, Integer, Boolean, Characters). Different kinds of oriented arc are distinguished (Figure 2). Three arcs represents the general domain model and two others instantiate the domain model according to a specific problem. The general structuring arcs are:

-the **att** arc links a concept and an **att**ribute concept of which the first one is composed,

-the **val** arc links two concepts where the first one takes its **val**ues in the second one

-the **spec** arc links two concepts where the first one is **spec**ialized (in the hierarchy sense) by the second one. This arc enables taxonomic hierarchy representation.

The instance description arcs are:

-the **elt** arc which links an element and its concept

-the = arc fixes the result value or entity of an attribute relationship.

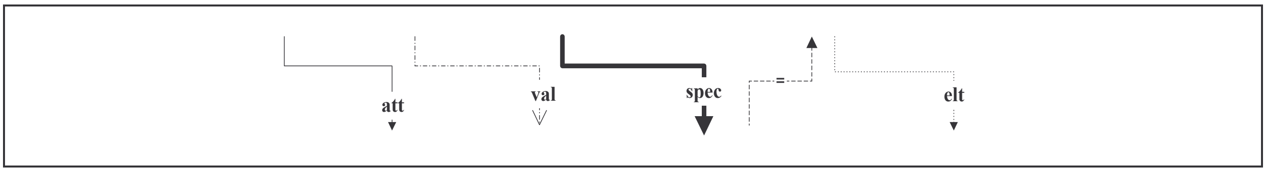

Figure 2:The semantic network arcs

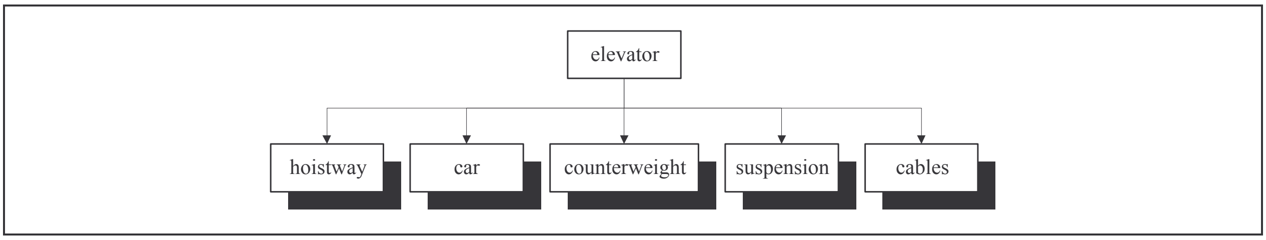

Figure 3: The *elevator* system net

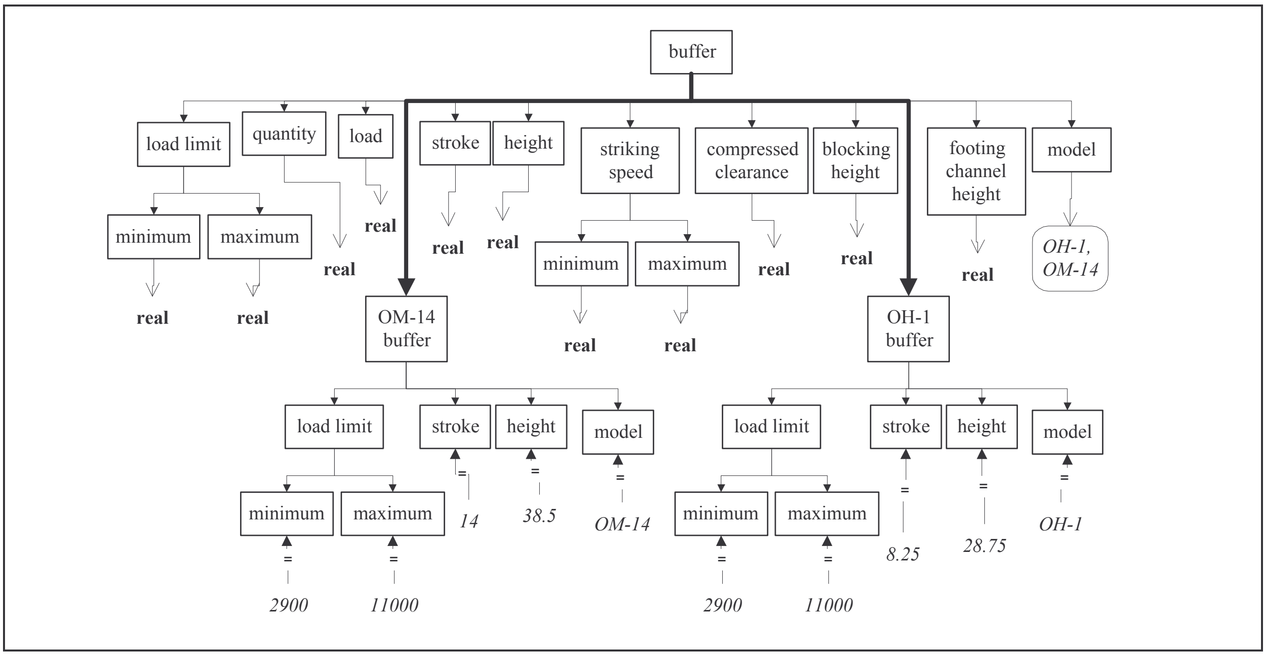

Figure 4: The *buffer* system net

After analyzing the Sisyphus II report, the elevator system composition can be deduced and represented in Figure 3 in an expressive manner. At first sight, it is understandable that an elevator system consists of five subsystem **att**ributes: the hoistway, the car assembly, the counterweight assembly, the suspension and the cables systems. Sisyphus II problem counts about 250 concepts.

We focus on the buffer system (Figure 4) which is a subsystem of the car and counterweight assemblies. A buffer model takes its **val**ues in the enumerated set {OH-1, OM-14} whereas a buffer height takes its ones in the real set. A buffer counts two sub**spec**ies, the OH-1 buffer and the OM-14 buffer, that are described by a specific model value (OH-1, res. OM-14), a specific stroke value (8.25, res. 14), a specific height value (28.75, res. 38.5), a specific load limit minimum (both 2900) and a specific load limit maximum (both 11000).

2.2 *Task language*

The task language is used to graphically represent dynamic knowledge. It has been defined taking into account some cognitive psychology studies applied to ergonomics [Scap89]. That is why a task, in MOISE, does not cover the same characteristics as a task in expert system studies [Chan86] and is much more elementary. A MOISE task offers no dynamic way to map data to roles and it owns a single predefined method. Two kinds of task are distinguished: *specific task* and *generic task*. A *specific task* is completely domain-dependent. It consists of input/output parameters (roles) and embodies subtasks (task structure) controlled by a generic task (structure control). A *generic task* is domain-independent. It only defines a generic control that may be used in the structure control description of specific tasks that share the same strategic control.

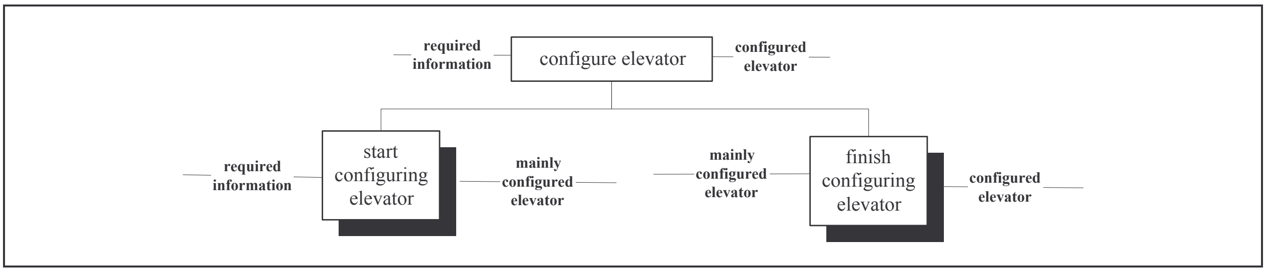

Figure 5: The *configure elevator* specific task

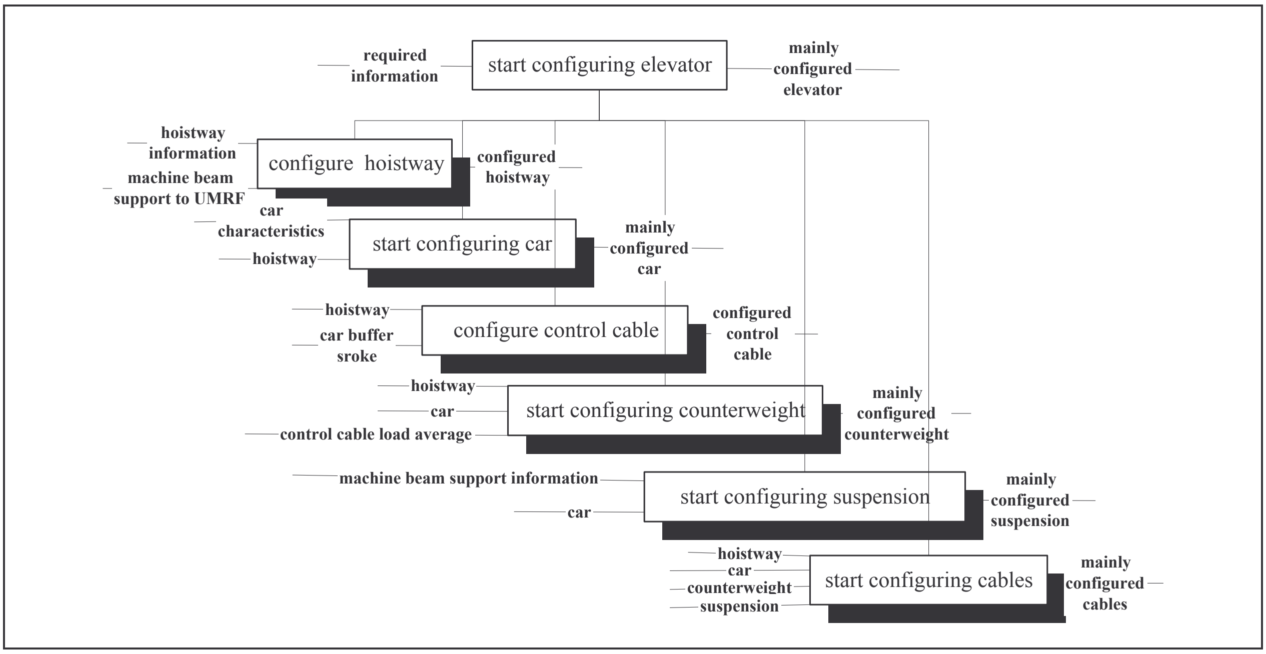

Figure 6: The *start configuring* specific task

The *configure elevator* specific task (Figure 5) consists of the input parameter named *required information*, the output parameter named *configured elevator* and the two subtasks *start configuring elevator* and *finish configuring elevator*. This indicates that to configure an elevator from the required information two steps are necessary (because of the data flow): you start configuring its components and then you finish configuring them. The *configure elevator* is the higher coarse-grained task and the more abstract one.

Figure 6 shows that, in order to start configuring an elevator system, you sequentially start configuring elevator subparts (Figure 3): you first configure the hoistway (completely), secondly the car (partly), thirdly the control cable (completely), fourthly the counterweight (partly), fifthly the suspension (partly) and sixly the cables (partly). At end you obtain the mainly configured elevator. If it is not indicated whether a task is specific or generic, the term *task* implicitly denotes a specific task, else the contrary is indicated.

For us, the fact is emphasized that dynamic knowledge specifications must map as accurately as possible to the proper reasoning way of the human expert. We could try to determine a generic task suitable for the expert by analyzing the Sisyphus II report. G.R. Yost writes to describe the derivation of a constrained value: « When deriving values for the design variables [...], process constraints as soon as possible. [...] The first step in processing a constraint is to see whether it is violated. If it is not, no further processing is needed for that constraint, and you can continue deriving values for design parameters or processing other constraints. If the constraint is violated, however, you should immediately try to find design modifications that remedy the violation. » We could sum up this fine-grained reasoning in the following manner (Figure 8). To derive a constrained value, a value is first proposed then a constraint is verified to see whether or not it is violated. If it is let's modify the right design parameters and do the different steps again (propose, verify, modify). Else let's stop.

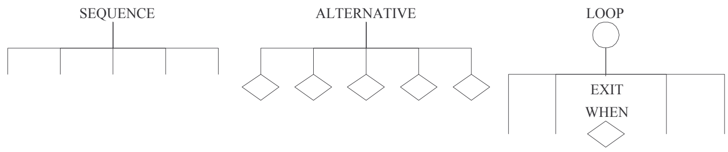

Figure 7: Three primitive generic tasks

A few primitive generic tasks are proposed Figure 7 such as *sequence*, *alternative* or *loop...exit when* generic tasks. These are so basic samples that the terminology *generic task* may seem too strong and *control task* should be more appropriate. But these very general control tasks may define the control of more precise generic tasks which get obvious generic features such as applicability and reusability features. For example the *loop...exit when* generic task is used to define the control of the *derive a constrained value* generic task which can then be refined in a specific task such as the *derive a car buffer load value* specific task (Figure 9).

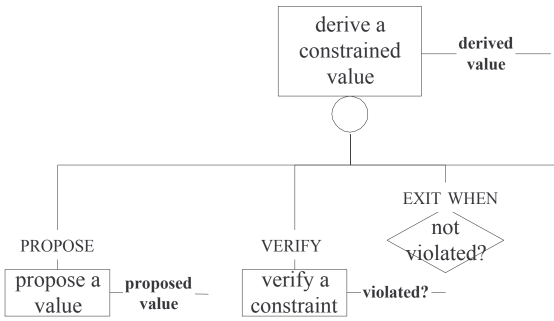

Figure 8: The *derive a constrained value* generic task

Graphical knowledge specifications are later converted in a textual form (Word document). This language keeps readability and understandability properties and proposes another way to describe specifications. Moreover its textual form implies that formal specifications are interpretable and computable by a syntactic analyzer that makes it easier to convert specifications in any programming language.

You can add to the formal specifications some new constraints you couldn't graphically represent. For example the following constraint mentioned in the Sisyphus II document : « Both car and counterweight assemblies are composed of several buffers and the buffer model must be the same for both car and counterweight assemblies in an elevator system » is written in the textual form:

elevator system ***att*** *car assembly* ***att*** *buffer* ***att*** *model = elevator system* ***att*** *counterweight assembly* ***att*** *buffer* ***att*** *model*

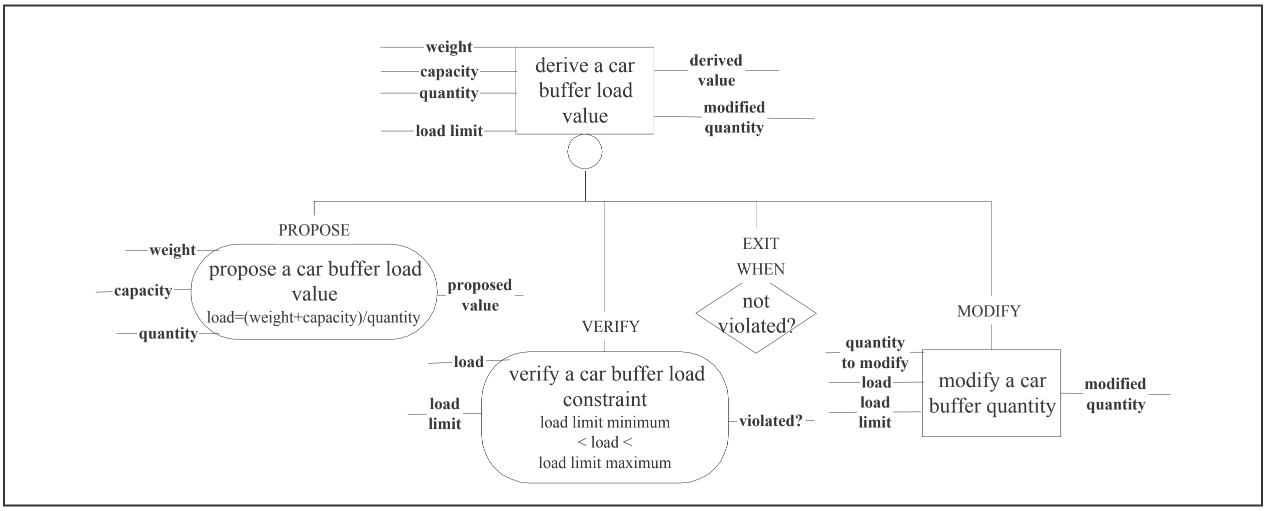

Figure 9: The *derive a car buffer load value* specific task

3. Object programming language

The last step (but not the least) to fulfil the operationalization purpose is to transform specifications from knowledge-level to symbol-level. We chose the high level programming language SPIRAL [Lorr92] [Lorr93] which is a logic object PL used for knowledge-based system development. SPIRAL is a meta-circular object-oriented language too [Coin89]. The design model that customizes the SPIRAL object model and enables static and dynamic knowledge transformation is described in this section.

3.1 Static objects

The predefined SPIRAL class model is nearly complete enough to describe net attributes and net constraints. A standard attribute class (*std_attribute*) is defined which consists of the following facets:

-the *domain* facet used to describe enumerated sets,

-the *value* facet used to fix a value,

-the *attribute* facet indicates if the attribute is *monovalue* or *multivalue*.

A *concept* is described with a *class*. Any class is created by the meta-class *class* and is a subclass of the class *object*.

3.2 dynamic objects

Three abstraction levels are available in SPIRAL model, they are the meta-class level, the class level and the instance level. A meta-class is a class that can create classes (meta-class instances are classes), and a simple class creates instances. These three levels are used to define task concept: *meta-tasks* at meta-class level, *task abstractions* (in short *tasks*) at class level and *task applications* at instance level are introduced:

meta-class	meta-task
class	task abstraction
instance	task application

Two meta-tasks are defined at the meta-class level : *meta-task* (or *specific meta-task*) and *generic meta-task*. The first meta-task consists of input and output parameters. The second meta-task inherits from the first one and consists of task parameters (plus inherited I/O parameters). These meta-tasks can be represented graphically with the net formalism in Figure 10.

The *specific* and *generic meta-tasks* can respectively create *specific* and *generic* tasks. From its input/output parameter values, the *specific meta-task* creates the proper I/O attributes of *specific* tasks. For example if input parameter names are (weight, capacity, quantity) and output parameter names are (proposed value), the *specific meta-task* creates a task with fields named weight, capacity, quantity and proposed value. From its input/output/task parameter values, the *generic meta-task* creates the proper I/O/T attributes of *generic* tasks.

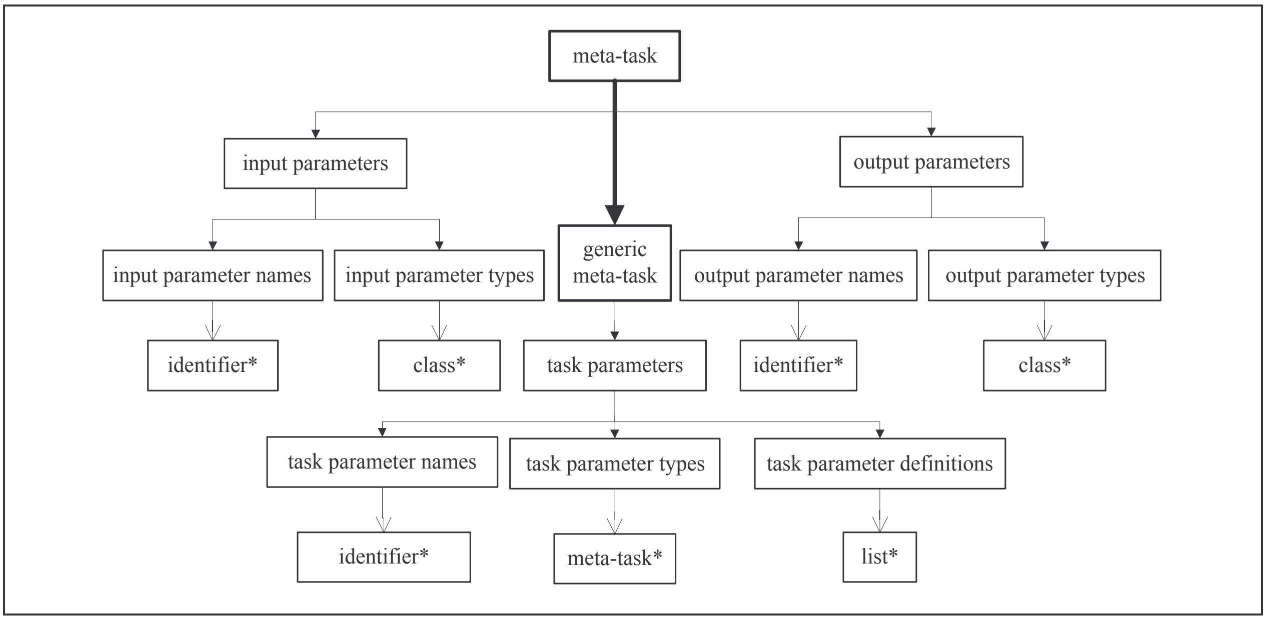

Figure 10: Meta-task net

3.3 Task taxonomy

The SPIRAL object-oriented language and its inheritance mechanism allow us to define object taxonomy. Concepts are modelled by classes so concept taxonomies (see buffer net) expressed in semantic nets are easily expressed in SPIRAL (*superclasses* field). Moreover tasks are modelled by classes so the SPIRAL inheritance property can be applied to tasks in a practical and useful purpose although no task taxonomy is expressed in the graphical task language. However task hierarchy definition presents a conceptual interest [Rech85] and tasks can be organized in a hierarchy. A task T' is more specific than a task T if T' parameter value domain is more restrictive than T one's and if both of them solve the same problem [Will94]. A task is described by I/O/T parameters and by a control defining how to execute the task. A task control is either a predefined algorithm or a *generic task application*. In the first case the task is *terminal* and in the second case it is *composed*. Six abstract tasks can be defined at the top of the task hierarchy and they depend on their parameter and control types. We therefore distinguish several tasks (Figure 11): *specific task (task), generic task, terminal specific task (terminal task), composed specific task (composed task), terminal generic task* and *composed generic task*.

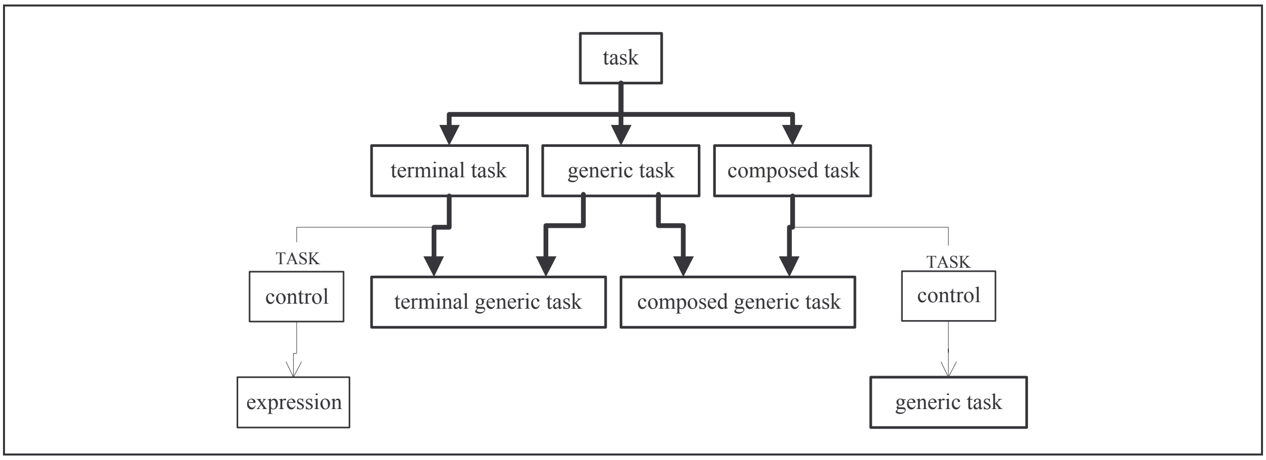

Figure 11: Task hierarchy top

The task hierarchy of the *derive a constrained value* task can be represented Figure 12 showing the conceptual interest of task hierarchy definition. The *derive a car buffer load value* task is more specific than the *derive a buffer load value* task that is more specific than the *derive a constrained value* task and they all have the same kind of goal.

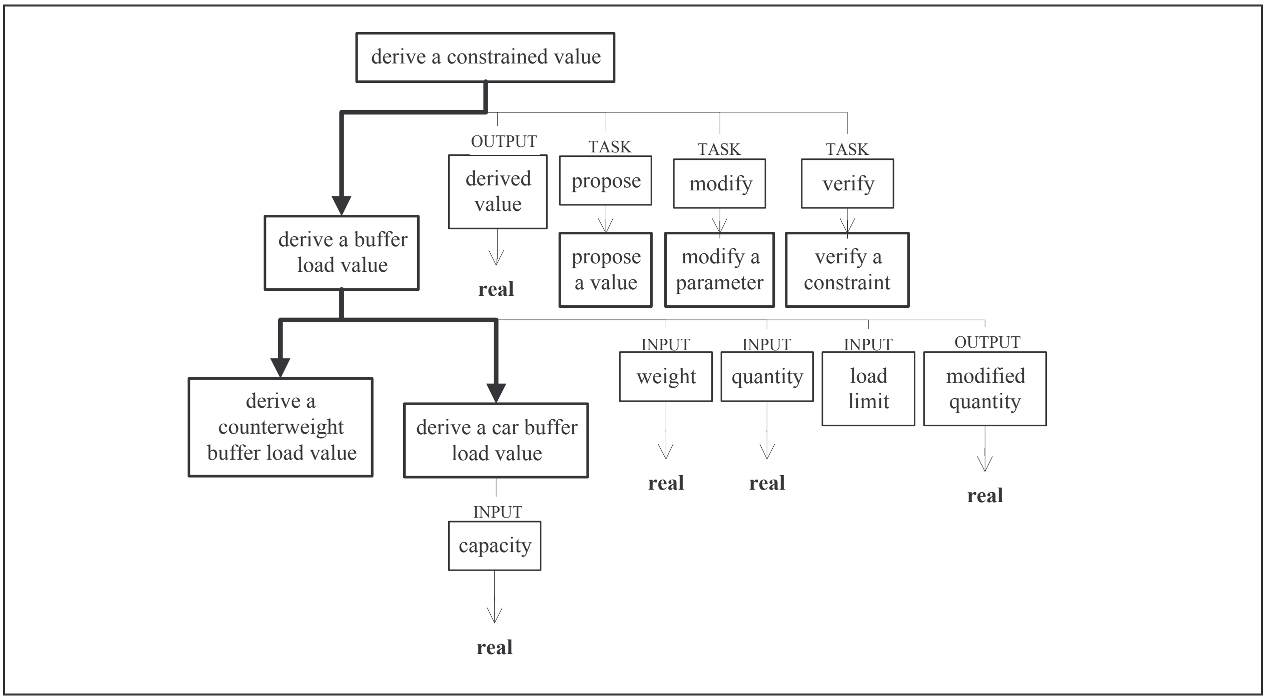

Figure 12: *Derive a constrained value* task hierarchy

Below are described in SPIRAL *derive a constrained value* composed generic task, and two different ways to define *derive a car buffer load value* task. The first solution does not use inheritance properties and enumerates the 4 input parameters (L1) and the 2 output parameters (L2) described on Figure 9. Its control applies directly the *derive a constrained value* generic task (L3) that apply the *propose a car buffer load value* (B1)*, verify a car buffer load constraint* (B2)*, modify a car buffer quantity* (B3) tasks through the *propose* (L4)*, verify* (L6)*, modify* (L8) task parameters. The second solution refines the *derive a buffer load value* generic task (L10) that refines the *derive a constrained value* generic task (L9). Because of inheritance properties, the *derive a car buffer load value* definition just need to contain a part of the I/O parameters (*capacity*) and a part of the control through *propose, verify, modify* task parameters (L11).

The description of the *derive a car buffer load value* task includes data flows that were not graphically represented for clarity reasons. For example the phrase L5 describes the data flow from the *proposed value* output parameter of the *propose a car buffer load value* subtask to the *derived parameter* output parameter of the *derive a car buffer load value* task. The phrase L7 describes the data flow from the *proposed value* output parameter of the *propose a car buffer load value* subtask to the *load* input parameter of the *verify a car buffer load constraint* subtask.

ask(generic meta-task, task create, ***derive a constrained value***
,(superclasses, composed generic task)
,(output parameter names, ***derived value****)*
,(output parameter types, real)
,(task parameter names, ***control*** *,****propose****,* ***verify*** *,****modify****)*
,(task parameter types, generic task, propose a value, verify a constraint, modify a parameter)
,(task parameter definitions
*,(****loop...exit when***
,(exit condition, violated COND)
,(task list,(propose, verify, exit when, modify)))
,()
,()
,()));

FIRST SOLUTION

```
    ask(meta-task, task create, derive a car buffer load value
        ,(superclasses, composed task)
L1      ,(input parameter names, weight, capacity, quantity, load limit)
        ,(input parameter types, real, real, real, load limit)
L2      ,(output parameter names, derived value, modified quantity)
        ,(output parameter types, real, real)
        ,(task parameter names, control)
        ,(task parameter types, generic task)
        ,(task parameter definitions
L3  ,(derive a constrained value
L4      ,(propose                                   B1
            ,(propose a car buffer load value
                ,(weight,(derive a car buffer load value, weight))
                ,(capacity,(derive a car buffer load value, capacity))
                ,(quantity
                    ,(derive a car buffer load value, quantity)
                    ,(modify a car buffer quantity, modified quantity))
L5              ,(proposed value,(derive a car buffer load value, derived value))
        ))
L6      ,(verify                                    B2
            ,(verify a car buffer load constraint
L7              ,(load,(propose a car buffer load value, proposed value))
                ,(load limit,(derive a car buffer load value, load limit))
        ))
L8      ,(modify                                    B3
            ,(modify a car buffer quantity
              ,(quantity to modify
                    ,(derive a car buffer load value, quantity)
                    ,(modify a car buffer quantity, modified quantity))
              ,(load,(propose a car buffer load value, proposed value))
                ,(load limit,(derive a car buffer load value, load limit))
              ,(modified quantity,(derive a car buffer load value, modified quantity))
        ))
    )));
```

SECOND SOLUTION

```
    ask(generic meta-task, task create, derive a buffer load value
L9      ,(superclasses, derive a constrained value)
        ,(input parameter names, weight ,quantity, load limit)
        ,(input parameter types, real, real, load limit)
        ,(output parameter names, modified quantity)
        ,(output parameter types, real));

    ask(generic meta-task, task create, derive a car buffer load value
L10     ,(superclasses, derive a buffer load value)
        ,(input parameter names ,capacity)
        ,(input parameter types, real)
L11     ,(task parameter names, propose, verify ,modify)
        ,(task parameter types, propose a value, verify a constraint, modify a parameter)
       ,(task parameter definitions

                    same as B1

                    same as B2

                    same as B3

       ));
```

3.4 *Task execution*

Any composed or terminal task class owns an *execution* method (in the object-oriented language meaning). The *execution* method running implies valuation of all task and output parameters. An *execution* running produces task applications (Figure 13) and each task application creates and controls its subtask applications. For example the *derive a car buffer load value#0* task application creates the *loop exit...when#0* task application that creates and controls the n *propose a car buffer load value#i*, the n *verify a car buffer load constraint#i*, the n *exit when#i* and the n-1 *modify a car buffer quantity#i* task applications. And so on.

A task application is an instance, so an *execution* is stored and will be eventually available for explanation use. For example the *YQT elevator car* may be configured many times but all the *configure car* task applications store the different car configuration cases. An *execution* browser has been developed that provides a navigation into task applications, that means into *execution* results.

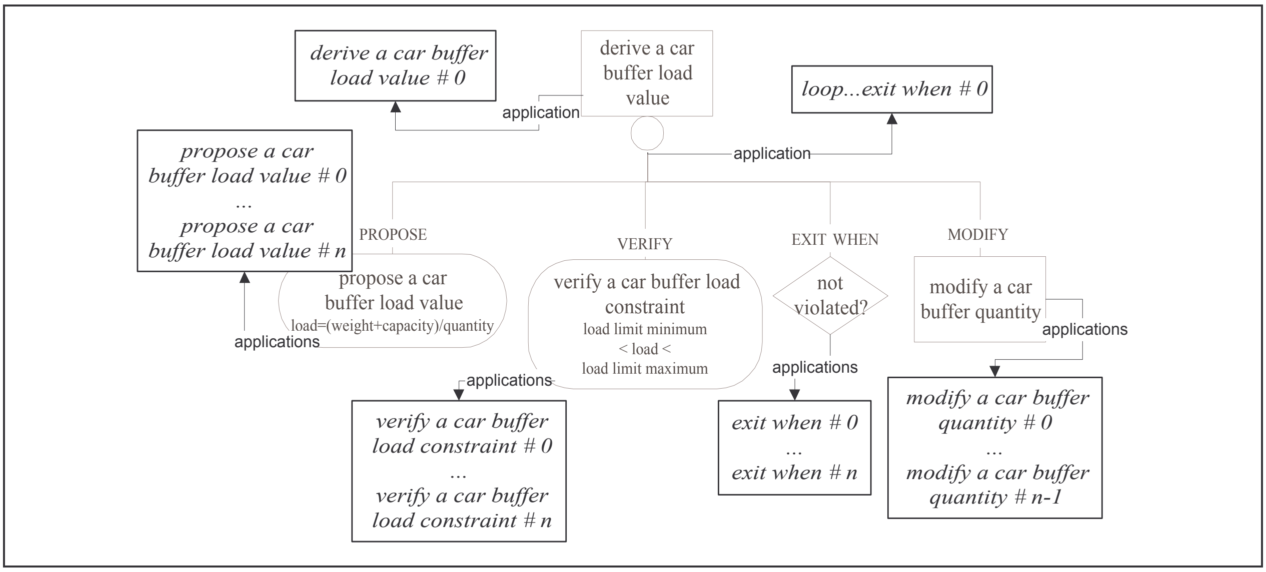

Figure 13: applications resulting from an execution

4. Conclusion

The aim of MOISE specification stage is not only to lead to an expert system that can solve a problem (configure an elevator system in Sisyphus example). It also aims at giving to an user an understandable description of the problem solving (of what an elevator is (Figure 3), of how a human elevator expert configures it (Figure 5, Figure 6) and therefore of how the expert system achieves the elevator configuration).

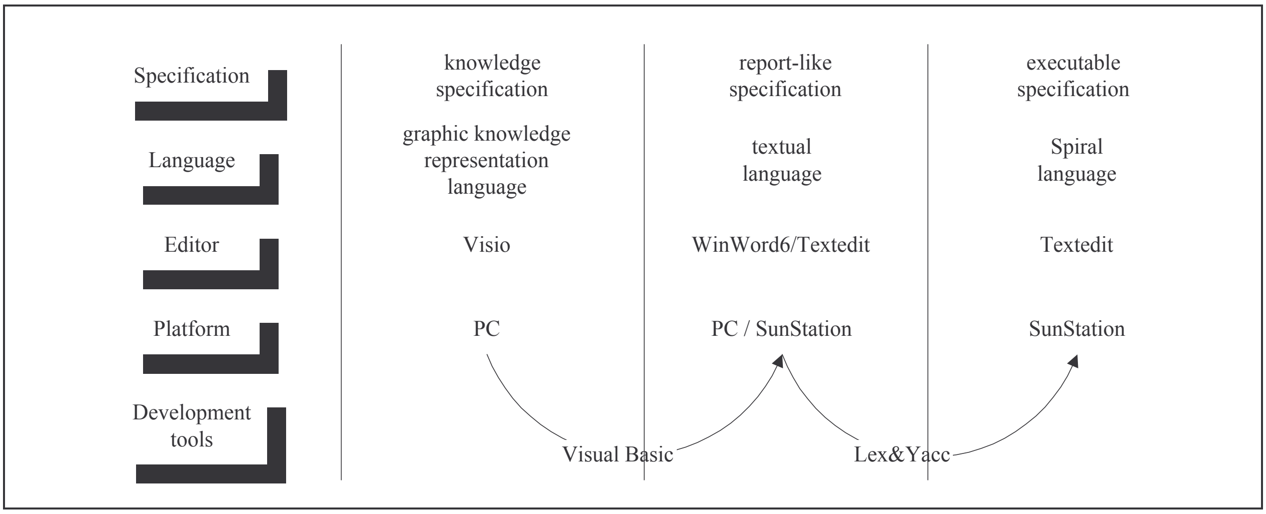

Figure 14: Operationalization steps

The main steps that lead to operationalize knowledge specifications are now defined and a design model, implemented in the SPIRAL object-oriented language, allows a smooth transition from knowledge-level to symbol-level. We are now working to automate the process of producing an executable specification from a formal model (Figure 14). A first tool that transforms graphical specifications in textual specifications and a second one that transforms textual specifications in SPIRAL specifications, are being developed. We are working at the same time to improve ergonomic features of task interfaces drawing our inspiration from ergonomic studies [Scap89]. We must notice that efficiency problems were not taken into account. We will keep on referring to the Sisyphus II problem which is a good way to test and evaluate our further works.

References

[Benz95] Z. Benzian, B. Bergeon, J-L Ermine, C.M. Falinower: "Knowledge Engineering for a Power Plant Operations Support System". LSS95 7th IFAC/IFORS/IMACS Symposium on Large Scale Systems: Theory and Applications, London (England), july 1995 .

[Bläs89] K.H. Bläsius, U. Hedstück, C.-R Rollinger (eds): *Sorts and Types in Artificial Intelligence*. Lecture Notes in Artificial Intelligence, Springer-Verlag ed, 1989.

[Breu94] J. Breuker, W. Van de Velde (eds): *Common KADS library for expertise modelling (reusable problem solving components)*. IOS Press ed, Amsterdam (Netherlands), 1994.

[Brun94] E. Brunet, J-L. Ermine: "Problématique de la gestion des connaissances des organisations". Ingénierie des systèmes d'information, AFCET-Hermes, vol 2, No 3, p 263-291, 1994.

[Chai95] M. Chaillot: " Conception d'une application hypermédia en gestion des connaissances pour la conduite des centrales nucléaires en mode dégradé.". Rapport interne SMTI/GGC/MC/95/026, 1995.

[Chan86] B. Chandrasekaran: "Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design". IEEE Expert, p23-30, aut 1986.

[Coin89] P. Cointe, J. Balder: "Metaclasses are first classes". the Objvlisp model, proceedings of ACM Oopsala 87, p 156-167, 1989.

[Derm88] J. Mac Dermott: "A taxonomy of problem solving methods". In Automating knowledge acquisition for Expert Systems, Marcus ed., p 225-256, 1988.

[Ermi91] J-L. Ermine: "Un système à base de connaissances pour le diagnostic technique et la recherche documentaire". Convention IA91, Hermes ed , Paris (france), p 241-256, 1991.

[Ermi93] J-L. Ermine: *Génie logiciel et génie cognitif pour les systèmes à base de connaissances*. Lavoisier ed, Tec et Doc, vol.1 et vol.2, 1993.

[Lorr92] J-P. Lorre, J-M Evrard, E. Dorlet: *Bases de connaissances pour la modélisation d'installations industrielles,* Journées d'Avignon, Avignon, France, juin 1992.

[Lorr93] J-P. Lorre: *SPIRAL: un langage objet logique,* Journées CEA-CUIC 93 (thème: l'approche objet: vers une nouvelle technologie de développement), Grenoble, France, juin 1993.

[Marc88] Marcus, Stout, and Mc Dermott: *VT: An expert elevator designer that uses knowledge-based backtracking,* AI magazine, spring 1988.

[Newe82] A. Newel: *The knowledge level,* AI magazine, vol 18, 1982.

[Peir78] C.S. Peirce: *How to make our ideas clear,* Popular Science Monthly, pp 286-302, January 1878.

[Rech85] F. Rechenmann: *SHIRKA:Mécanismes d'inférence sur une base de connaissances centrée-objet,* 5e congrès Reconnaissance de Formes et Intelligence Artificielle, p 1243-1254, 1985.

[Scap89] D.L. Scapin, C. Pierret-Goldbreich: "Towards a method for task description: MAD". Proceedings of Work with display units conference, Montreal (Canada), 1989.

[Sowa91] F. Sowa: *Principles of semantic networks*: J.F. Sowa ed, Morgan Kaufmann , 1991.

[Stee89] L. Steels: "Components of expertise". Rapport VUB AI Memo 89-2, Artificial Intelligence Laboratory, Université de Bruxelles (Belgique) 1989.

[Voge89] C. Vogel: *Génie cognitif*. Masson éd., 1989.

[Wiel92] B.J. Wielinga, A.Th. Schreiber, J.A Breuker: "KADS: a modelling approach to knowledge engineering". Proceedings of Knowledge Acquisition, vol 4, No 1, p 5-53, march 1992.

[Will94] J. Willamovski: *Modélisation de tâches pour la résolution coopérative de problèmes,* Thèse, LIFIA, Grenoble (France), 1994.

[Yost92] Gregg R. Yost: *Configuring elevator systems,* Technical report, Digital equipment Co, Malboro, Massachussetts, 1992.